JSTyper: Type inference for JavaScript

Christopher Little\textsuperscript{2,1} Kathryn E. Gray\textsuperscript{2} Scott Owens\textsuperscript{1}
\textsuperscript{1} University of Kent\textsuperscript{1} \textsuperscript{2} University of Cambridge

Abstract
As with all untyped languages, JavaScript programs can contain type-mismatch-style errors. These mistakes do not necessarily cause runtime errors and can lead to subtle faults in deployed web apps. Type checking key portions of JavaScript applications can identify these mistakes early. Previous efforts to augment JavaScript with a type system have led to syntax extensions and compulsory type annotations; these increase the difficulty of integrating type-checked code into a significant code base.

In JSTyper, we provide a type system which supports a broad range of idiomatic JavaScript programming, including dynamic property addition and object prototyping, and does not require type annotations. We use gradual typing-style checks to integrate with untyped JavaScript. To accomplish this, we separate a program into typed and untyped portions, checking the former and using dynamic checks to provide safe interaction with the latter. This in turn provides a low-effort migration path to checked JavaScript programs.

We present our type system, a proof of soundness, and a practical implementation written in JavaScript that can be run in any modern browser.

1. Introduction
JavaScript is a high-level, dynamically-typed programming language supporting both functional and object-oriented programming idioms. Although originally designed for little more than validating forms on websites, today the language is an essential part of everyday activity on the internet, powering demanding online applications such as image editors, IDEs, online games, and e-commerce. As the scale of JavaScript applications has increased, so too has the complexity of development projects. Assurance that a program is bug-free has become more critical.

JavaScript’s weak typing discipline combined with its type coercions, however, make it hard to reason about a program’s correctness, and limit the information available to development tools. With coercions many expressions that would normally be deemed untypable or raise an exception instead convert an unexpected value to an acceptable one, for example 4 to \texttt{true} when placed in a context expecting a boolean. Additionally, attempts to access missing object fields results in a special \texttt{undefined} value, which can then be coerced. This can result in the silent propagation of type errors, providing no explicit errors to the developer.

We adopt a gradual typing approach to integrate a static type system, supporting many common idioms, with genuinely unty- pable code. Typable idioms include the dynamic addition of properties (fields) to objects, creation from a prototype, change of prototype, and higher-order functions. Our type inference does not require any annotations to check these idioms, which is a significant contribution of this work.

Related work
ECMAScript 4 \cite{ECMA-262}, a proposed extension to the JavaScript standard, featured optional type annotations, but the proposal was never finalised and the annotations are not present in later versions of the standard. TypeScript \cite{Manning2012}, uses a superset of JavaScript syntax with similar annotations, and compiles this back into JavaScript. These approaches have depended on programmer-inserted type annotations, which limit their utility for legacy codebases. Some projects have attempted to use Hindley-Milner style analysis to infer the types of program expressions automatically \cite{Henderson2007, Kelsey2015, Pimentel2012}, including the most recent contribution, Flow \cite{Flow}. These efforts, however, have fallen back on using annotations to resolve parametricity in object types where the system too tightly restricted bounds on members, and in specifying class types and signatures. While this expressibility may be desired by some users of a type system over JavaScript as a means of specifying and enforcing a particular view of the JavaScript objects, it does require that source code is modified.

Any type system, however, will struggle to offer reliable results because of the inherently dynamic nature of the language. Many traditional JavaScript programming idioms are legitimately untypable. JavaScript has no mechanism for overloading functions, for example, so many developers write functions expecting different parameter types from call-to-call, and use manual type checks to handle them (see Listing 1). Flow and TypeScript allow for untypable functions using a special ‘any’ type, which will never trigger a static type error, but these do not provide gradual type style guarding to trigger runtime errors instead. The disadvantage here is that the system can no longer make any guarantees about the soundness of the program, as certain parts of it are not checked. With gradual typing, we provide the assurance that the typed portions of a program are safe and that any runtime errors are caught before invalidating the typed code, with the untyped portions of the program to blame.

Additionally, JavaScript code rarely operates in isolation in modern web applications. Even if a program passes the static type checker, some external piece of code may introduce a type error by, for example, passing data of the incorrect type to a well-typed function.

Gradual typing is a technique used to make safety guarantees about code which combines both static and dynamic typing. Initially looking at interoperation between static and dynamic languages \cite{Liang2003, Wegner2003, Niederhauser2007}, as well as adding untyped parameters to Java programs, the concepts are also applicable here, where the division
Listing 1: Dynamic idioms in JavaScript

between static and dynamic code lies within one language rather than between several. Since static analysis cannot guarantee that the program is free of dynamic type errors, we instead guarantee that only dynamic properties of the code are able to cause runtime errors – that the well-typed code “can’t be blamed” [17] for any type errors which arise. This is achieved by protecting the boundaries between the type-safe and type-unsafe ‘worlds’. Conversions are made from the dynamic values, and although this conversion process may generate an error (for example trying to convert a number to a string), the type safety of the well-typed world is provably preserved.

Contents In this paper, we present a type system for JavaScript (§2) along with a corresponding operational semantics, and prove that the system defined has the properties of progress and type preservation (§5). §3 presents the type inference implementation, including a source-to-source compiler which augments programs with runtime checks for interacting with untyped JavaScript, and describes preliminary results of this type checker over programs from the SunSpider[12] benchmark suite.

2. Type system

Despite JavaScript’s dynamic flexibility, within one function we believe it is rare for a programmer to intentionally use the same variable with differing incompatible types. However, programs do dynamically grow the members of an object and change the inheritance prototype chain. Factoring in these constraints, we have designed a type system that permits object growth via adding properties, prototyping, or changing prototype but that infers a consistent type for variables within one function.

In the remainder of this section, we provide the JSType type rules and emblematic programs that illustrate how these rules work and why these choices have been made.

2.1 Formal framework

Our type judgements are divided into typing judgements for expressions and typability judgements for statements. A typing judgement has the form

\[ \Gamma \vdash e : T \mid \Gamma' \]

and a typability judgement has the form

\[ \Gamma \vdash m \mid \Gamma' \]

\(\Gamma\) represents the type store by providing a map from variable identifiers to types. \(e\) is the expression being judged, and \(T\) is a valid type for this expression. Statements \(m\) do not themselves have a type, so we use a typability judgement of the same form without \(T\). The typability judgement asserts that all sub-expressions of the statement are well-typed. \(C\) is a set of constraints which must be satisfied for the type judgement to be considered valid.

Finally, evaluating the expression may result in changes to the type store (for example introducing a new variable), and these changes are reflected in \(\Gamma'\), which will be used for judging the types of subsequent expressions.

\[ T \text{ is fresh: } \Gamma \vdash e : T \mid \Gamma_1 \]

By way of example, rule Proptype above indicates that \(e\) only has a valid type if \(e\) does, and if the type of \(e\) is a subtype of the type \([l : T]\) for some \(T\) (i.e. if the type of \(e\) contains at least the property \(l\)).

2.2 Forms of Constraint

The constraints generated in the type judgements may take one of three forms – equality, subtype or optional constraints.

An equality constraint \((T_1 = T_2)\) is the most straightforward, and simply indicates that two types are equal. A subtype constraint \((T_1 \geq T_2)\) indicates the same thing for primitive types, but for object types indicates that every property present in the supertype \((T_1)\) must also be present in the subtype \((T_2)\). Furthermore, the equivalent properties in the two types must obey the same relation. Formally:

\[ T_1 \geq T_2 \iff \forall [l : T] \in T_1. \ (l : T') \in T_2 \land T \geq T' \] (1)

Finally, an optional constraint \((T_1 \geq_o T_2)\) is slightly weaker. It makes no requirement that every property of \(T_1\) be present in \(T_2\), but properties’ types must be related if they are present in both. Formally:

\[ T_1 \geq_o T_2 \iff \forall [l : T] \in T_1. \ (l : T') \in T_2 \Rightarrow T \geq T' \] (2)

2.3 Object Growth

It is permitted (and common) in JavaScript to add properties to an object after its creation. Some approaches to typing JavaScript make the assumption that all objects have a larger ‘potential’ type than they are created with, and that uninitialised properties will be added during some extended initialisation phase [2]. Analysis of JavaScript’s dynamic behaviour in practice, however, suggests that adding properties is likely to happen at any point in the object’s life cycle [14], and so potential types are not a good model.

We instead generate a ‘growth effect’ any time a property assignment is made. It is made up of three components: the type of the object whose property we are assigning to; the name of the property we are assigning to; and the new type of the property. For example, after witnessing the assignment \(a.b = \text{true}\), we generate the effect

\(T_a, b, \text{boolean}\).

apply is used to add the new property to all instances of \(T_a\) within the type environment. The apply function achieves this safely with a simple substitution in most cases. If \(T_a\) appears as the type of an object property, however, it instead generates a new growth effect, which is recursively applied again to the resulting type environment. We also create an optional constraint, ensuring both that \(T_a\) is an object type, and that, if \(b\) is not a novel property, then the previous property also had type \text{boolean}. This is all represented in rule ProptAssignType.

We cannot simply add the novel property to \(T'\) directly, since we need to be able to distinguish pre-existing properties from novel ones introduced within a scope. Additionally, \(T'\) may already be
constrained as a supertype of another type, and adding properties to $T'$ would violate this constraint. Instead, one must consider two separate object types— one where the new property is present, and one where it is not. At the start of the function, $x$ should have type

$$T_s = \{b : \{c : \text{number} \}\}.$$  

otherwise line 3 would involve an undefined property access. At the end of the function, after adding the property $c_2$, $x$ should have type

$$T'_s = \{b : \{c : \text{number}, c_2 : \text{bool} \}\}.$$  

When determining the type of $f$, we must have access to both of these types in order to construct the correct function type:

$$T_s \rightarrow T'_s = \{b : \{c : \text{number} \}\} \rightarrow \{b : \{c : \text{number}, c_2 : \text{bool} \}\}.$$  

We cannot simply replace all occurrences of $T_s$ in the type environment, as this would result in the type of $f$ becoming $T'_s \rightarrow T'_s$. It is also insufficient to simply set

$$\Gamma' = \Gamma \cup \{x : T'_s\}$$  

(which would leave other occurrences of $T_s$ in $\Gamma$ unchanged). The property access on line 3 will generate constraints of the form

$$(b : T_s) \geq T'_s$$

$$(c : \text{number}) \geq T_h$$

The problem is that these only constrain $T'_s, T_h$ is left essentially unconstrained, and $f$ would be incorrectly given the type

$$(\{b : \} \Rightarrow \{b : \{c : \text{number}, c_2 : \text{bool} \}\})$$

In order to correctly judge the type of this function, we need constraint (3) to pass through $T'_h$ to $T'_s$. Our effect achieves this by attaching $T_s$ as the origin of $T'_h$. We would then like to replace all references to $T_s$ in the type environment to which the effect is applied with this new derived object, represented by $T'_h \Rightarrow T'_s$. The $\text{apply}$ function achieves this with either a straight substitution from $T_s$ to $T'_h \Rightarrow T'_s$ or, in the case of object properties, by generating new effects and recursively applying these. We must give a definition for subtype and optional constraints involving such a derived type. When a derived object type (i.e. one which has an origin defined) is constrained as a subtype, it may either have the required properties itself, or find them further down the origin chain. Formally, we can partition the properties of the supertype $T_i$ into properties which can immediately be found in $T_j$, and properties which can be found further down the chain (i.e. somewhere in $T'_j$):

$$T_i \geq T_j \Rightarrow T'_j \iff \exists T_{ao}, T_b, \forall \{l : T\} \in T_{ao}, \forall \{l : T'\} \in T_b, (\{l : T\} \in T_2 \land T \geq T') \land T_b \geq T_2$$

$$\land T_{ao} \cup T_b = T_1$$

(4)

and

$$T_1 \geq_\omega T_2 \Rightarrow T'_2 \iff \forall \{l : T\} \in T_{ao}, \forall \{l : T'\} \in T_2 \Rightarrow T \geq T'$$

$$\land T_{ao} \geq_\omega T_2.$$  

In the other direction, we wish to ensure that all properties defined in the supertype’s chain can be found in the subtype:

$$T_1 \Rightarrow T'_1 \geq T_2 \iff T_1 \geq T_2 \land T'_1 \geq T_2$$

(6)

and

$$T_1 \Rightarrow T'_1 \geq_\omega T_2 \iff T_1 \geq_\omega T_2 \land T'_1 \geq_\omega T_2.$$  

This system allows constraints to propagate along the origin chain, finally being enforced on the root object.

1. function $f(x)$
2. x.b.c2 = true;
3. return x;

Listing 2: An example property addition

1. \textbf{if} (...) {
2. x.a = true;
3. }
4. x.a = 5;

It is also worthwhile considering the properties which may have been defined on an object type. Although these properties may
never safely be read, we may also wish to enforce that subsequent writes to the property have a consistent type. In Listing 2.3 for example, the write on line 4 would not cause any particular type safety issues (since the boolean definition of \(x.a\) is effectively invisible at this point). Nonetheless, we interpret such flexibility with property types as a sign of a programmer error which should cause a type error. To achieve this, we also add all property types which were not in the intersection of the two chains, but we wrap these types with an ‘IllDefined’ wrapper type. This will trigger an error if the property is ever read, but also gives a type for subsequent property reads to unify with. This behaviour is important for typing return statements.

2.4 Return

\[
\Gamma \vdash \text{return } \text{undefined} \mid \Gamma \quad \text{RetTypable1} \\
\Gamma \vdash e : \Gamma \quad \Gamma \vdash \text{return } e : \Gamma' \\
\Gamma \vdash \text{"return"} = \text{pending} \\
\Gamma'[[\text{"this"}]] = \Gamma_c' \\
\Gamma'[[\text{\@const}\text{r}]] = \Gamma_c \\
\Gamma' = \Pi \text{all future types return } \Gamma' \\
\Gamma = \Pi \text{the IllDefined wrapper type. This will} \\
\text{trigger an error if the property is ever read, but also gives a} \\
\text{type for subsequent property reads to unify with. This behaviour} \\
\text{is important for typing return statements.}
\]

When a function has multiple return statements, we need to ensure that every possible return value returned has the same type. This parallels the situation with variable assignments, where we need to ensure that each assignment to a particular variable has the same type. We thus use the same mechanism for tracking return values as for variables, and include a special “return” entry in the type environment. Since every function may be used as a constructor, and the type of the object constructed is not necessarily the same as the return type of the function, we must also ensure that the type of this is consistent at every point the function could terminate. We track the type of this from one return statement to the next using a special entry “\@const\text{r}” in a similar way.

The absence of a return statement in a function indicates a function which effectively returns the value undefined. We cannot, however, simply initialise \(\Gamma[\text{"return"}]\) to the type undefined, since this would fail to unify with the return types of later return statements. Instead, \(\Gamma[\text{"return"}]\) is initialised to a special type pending when a function definition is encountered. Rule RetTypable2 handles the first return statement encountered by simply overwriting the pending entry in the type environment. RetTypable4 constrains all future return types to be compatible with the previous one. If \(\Gamma[\text{"return"}]\) is still pending after typing the whole function body, then the function is typed as having return type undefined (rules AnonVoid and NamedVoid).

Matters are complicated slightly in the presence of non-trivial control flow. The function below, for example, should not be typeable because it may sometimes return a number, but sometimes it will return undefined.

```plaintext
1 function isPositive(x) {
2   if (x>0) {
3     return true;
4   }
5 }
```

Because there is no explicit ‘return undefined;’ statement at the end of the function, an unsatisfiable constraint is not generated. This problem requires some sort of control flow analysis to solve. As part of the merge operation of rules In, While and ForTypable, type environment entries are given an ‘IllDefined’ wrapper type if they are well-defined in one branch but not in the other. In particular, this operation will apply to the return entry. If the return type is ill-defined when a new return statement is encountered, the two types will be constrained (as for RetTypable4), and the return type will be well-defined in the outgoing type environment (RetTypable3). In all cases the value of this is saved as the constructor type. Provided it is consistent with previous values. If the return type is still ill-defined after typing the function body, then no type judgement is applicable (rules AnonFun and NameFun both require that \(\Gamma[\text{"return"}] \neq \text{IllDefined}(T)\) for any type \(T\), and the function is deemed unparsable. At this level, we can also determine whether the return type of the function is a primitive type. If it is not, then the ultimate constructor type of the function will actually be the same as the return type for the function.

2.5 Functions

\(T_0, T_1, \ldots, T_n\) are fresh

\(T_j = (\{\text{call} : \{T_0, T_1, \ldots, T_n \rightarrow (T_0, T_1)\}, \text{prototype} : \{\} \})\)

\(\Gamma = \Gamma \cup \{\text{this} : T_0, x_1 : T_1, \ldots, x_n : T_n, \text{return : pending, \@const}r : \{\}\})\)

\(\Gamma = \Gamma \vdash e : \Gamma_c \quad \Gamma'[[\text{"return"}]] = \text{pending} \\
\Gamma'[[\text{"this"}]] = \Gamma_c \\
\Gamma'[[\text{\@const}\text{r}]] = \Gamma_c \\
\Gamma' = \Pi \text{all future types return } \Gamma' \\
\Gamma = \Pi \text{the IllDefined wrapper type. This will} \\
\text{trigger an error if the property is ever read, but also gives a} \\
\text{type for subsequent property reads to unify with. This behaviour} \\
\text{is important for typing return statements.}
\]

To determine the type of a function, we need to create a new type environment and use it to type-check the body. The new environment will be based on the current one, but with additional entries for the parameter types. The return type is handled as discussed in the previous subsection (only rule AnonVoid is shown here but others are similar). Once the parameters and return type have been determined, we can create a type for the function. Note that the returned type of the function, \(T_0\), is an object with property @call rather than the closure type directly. This models the fact
that all functions are still objects, in JavaScript, and can legally be manipulated as such.

The most important such manipulation involves the .prototype property, which defines the properties inherited when the function is used as a constructor. In this case (rule ConstType), the prototype object type is attached as the origin of a new, empty object type to use as the initial type of this within the function. This mimics the behaviour of JavaScript itself, where a function’s .prototype property is implicitly attached to this. The effect of this is that dynamic modifications to the function’s prototype become visible to all objects constructed by the function, which will have its prototype as the base of their origin chain.

\[
\text{dom}(\gamma) \subseteq \text{dom}(s)
\]

\[
\begin{align*}
\gamma & \vdash \text{func} : [T, \gamma]_c \gamma' \\
\Gamma & \vdash [\text{func}, s] : [T, \gamma]_c \Gamma
\end{align*}
\]

\[
\text{V}_\text{CLOSURE}
\]

The typing rules for function closures involve capturing the type environment within the closure type. The restriction of \(\gamma\)'s domain is necessary to show that func will be well-typed when using s as scope. Note that expressions enclosed in \([ ]\) will never appear in the original program, and so the type judgements for these kinds of expression are required for the proof of type safety, but do not appear in the executable type inference system.

When using the closure’s type environment for a function call, we must first assert that the captured store is well-typed under this type environment (\(\gamma \vdash (s, \theta)\)). In essence, this tells us that the type environment \(\gamma\) and the store \((s, \theta)\) agree on the types of variables.

### 3. Implementation

The JSTyper implementation both verifies that the source code is well-typed and also rewrites the source program to insert run-time checks constraining the interaction between typed and untyped code, following the techniques of gradual typing. Because this is source-to-source, our compiled JavaScript can run in any implementation. We also have explored a more live-integration using node.js. Future work will include techniques to make the type checker and compiler implementation reachable from general browser specifications.

There are two parts to verifying that an individual typed program is correct: generating constraints according to the typing rules defined, and solving these constraints. If the constraints are satisfiable, then the program is well-typed. Although all that is required for type-safety is this satisfiability, we are also interested in finding a canonical type for each expression, as an aid for development tools.

#### 3.1 Constraint Generation

Our implementation generates constraints by precisely following the rules of the formally specified type judgements. UglifyJS\(^3\) is used to parse the input code and creates an abstract syntax tree. The tree is traversed node by node, in order to perform syntax-directed type checks following our typing rules.

#### 3.2 Constraint Solution

Pierce\(^\(11\)\) presents a unification algorithm based on Hindley and Milner’s ideas to calculate a solution to a set of equality, rather than subtype constraints. The algorithm is simple and operates in linear time in the number of constraints (which is itself linear with respect to the program size). It operates by iteratively turning constraints involving abstract type variables into substitutions, such that each iteration eliminates one abstract type variable. This is made possible by the fact that the constraints are equations – if \(T_1 = T_2\), then it is always possible to substitute \(T_1\) for \(T_2\). If a constraint is found which equates two different concrete types, the algorithm fails. A side-effect of this algorithm is finding the most general unifier for each type variable, which gives a canonical representation of the type. We have generated subtype rather than equality constraints, however, which are not so easily solved.

We cannot simply use substitution in the same way for subtype constraints, because the two constrained types may not be equal. Even with an algorithm to determine satisfiability, we cannot find a single most general unifier for object types. The most precisely we can hope to pin down a type variable to is within an upper and lower bound. In listing\((3)\)\(\) for example, \(x\) must have the property \(c\), but \(b\) and a may or may not be present.

Listing 3: An example of a program with multiple type solutions

```plaintext
1  x = {a: 1, b: 2, c: 3}
2  // T_2 \geq \{a: number, b: number, c: number\}
3  // T_2 \leq \{c: number\}
```

Although these problems are introduced by subtyping constraints, in practice they will only arise for object types. Primitive types like numbers or strings, for example, lack any kind of substructure and so a subtype constraint involving a primitive type can essentially be treated as an equality constraint. Function and array types do have substructure, but solving a subtype constraint for these types simply involves passing on the constraint to a lower level of structure. Table 1 summarises these solution methods. Since most types can be handled easily using substitution, it would be nice to find a solution for objects and hence retain the benefits of the (linear-time) unification algorithm. The following section discusses our attempt to do this.

#### 3.3 Object Subtype Constraints

Let \(A\) be an abstract type, and \(O\) be a concrete object type. For the constraint \(O \geq A\) to be satisfied, \(A\) must be an object type with at least all the properties of \(O\). If some property were present in \(O\) but not in \(A\), then \(A\) would not be a valid subtype of \(O\). So for this constraint it is reasonable to use the substitution \([O'/A]\), where \(O'\) is a clone of \(O\). The reason we do not use \(O\) directly is that later constraints may add more properties to \(O'\), but these should not have an effect on \(O\).

For the inverse constraint, \(A \geq O\), the substitution is perhaps not so obvious. Although the same substitution \([O'/A]\) does satisfy the constraint, by using it we are rejecting any object types smaller than \(O\), which would also satisfy the constraint. This approximation means that some potentially legal programs will be rejected by the implementation. Of course, static type-checking is inherently conservative. Any type checker which is sound and decidable must be incomplete – that is, it will always be possible to write a program which is well typed but does not satisfy the type checker. The approximate substitution used here maintains type safety – any program which satisfies it must be well typed – at the cost of a reduction in completeness. Empirical observation through testing fragments of JavaScript has not revealed any ‘real’ programs which suffer as a result of this approximation, however.

The last case to consider is a constraint between two object types, \(O_1 \geq O_2\). We can check whether the constraint is satisfied by

---

\(^3\) UglifyJS. The JSTyper implementation both verifies that the source code is well-typed and also rewrites the source program to insert run-time checks constraining the interaction between typed and untyped code, following the techniques of gradual typing. Because this is source-to-source, our compiled JavaScript can run in any implementation. We also have explored a more live-integration using node.js. Future work will include techniques to make the type checker and compiler implementation reachable from general browser specifications.
inspection – iterate through all properties of \( O_1 \) and ensure that they are present in \( O_2 \). If all are present, we create subconstraints for the corresponding properties of both objects and move on. If some property is missing, however, a conservative type checker must deem the program untypeable. Although this system is provably sound, it is very much incomplete, and most interesting programs will fail under it.

The implementation of a more suitable constraint solver will involve compromise between completeness and complexity. The results proven about the specification (progress and type preservation) rely on the assumption that the constraints generated are satisfiable, and their correctness is thus independent of the particular approach taken to determine their satisfiability. The practical approach we have taken is described below, though further investigation into its formal aspects may be warranted. An alternative strategy may involve keeping explicit track of upper and lower bounds for each type, and collapsing these bounds down to provide a canonical type for the variable. Preliminary investigation into the work of François Pottier \([13]\) indicates that his approach may also be of use here.

Our implementation involves the gradual ‘discovery’ of new properties. In certain contexts, it would be safe to add properties to \( O_2 \) in order to make the constraint satisfiable. An example of one such context is shown in Listing 4. The types of function parameters and imported variables are suitable for this kind of discovery, since we do not have an initial assignment of an object which would normally provide a reasonable approximation of the object type. Our implementation does exactly this – when initialising a type for a function parameter, a shouldInfer flag is set which will allow discovery of new properties when the type is involved in a constraint. After the new properties are discovered, they will be reflected in the function type – and if the function is called with values lacking these properties then the type checker will fail.

Listing 4: A context in which discovery of new properties is safe

```
function f(x) {
  x.foo += 1;
  // T, ≤ ({foo: number}, solved by substitution
  return x.bar;
  // T, ≤ (bar: T), solved by discovery
}
```

<table>
<thead>
<tr>
<th>Primitive T</th>
<th>T = ([T_1])</th>
<th>T = ((T_1, \ldots, T_r))</th>
<th>Abstract T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive T’</td>
<td>Fail if ( T \neq T’ )</td>
<td>Fail</td>
<td>Fail</td>
</tr>
<tr>
<td>( T' = [T_1] )</td>
<td>Fail</td>
<td>( T_1 \geq T_r )</td>
<td>Fail</td>
</tr>
<tr>
<td>( T' = (T_1, \ldots, T_r) )</td>
<td>Fail</td>
<td>Fail</td>
<td>( T_r \geq T_1, \ldots, T_r \geq T_i )</td>
</tr>
<tr>
<td>Abstract T’</td>
<td>( [T/T'] )</td>
<td>( [T/T'] )</td>
<td>( [T/T'] )</td>
</tr>
</tbody>
</table>

Table 1. Simple substitution and subconstraint cases for the constraint \( T \geq T' \)

For all \( i : T_i \) in \( O_1 \), we must have some \( i : T_j \) in \( O_2 \). We can simply push the constraint down, generating the new constraints \( T_i \geq T_j \). Array types are similar: for a constraint \( [T_1] \geq [T_2] \), we simply generate the new constraint \( T_1 \geq T_2 \). Finally, functions behave in a similar way with regards to their return type, but are contravariant in their argument types. Hence for a constraint \( (T_1, \ldots, T_r \rightarrow T_j) \geq (T_1', \ldots, T_r' \rightarrow T_j') \), we generate the new constraints \( T_1 \leq T_1', \ldots, T_r \leq T_r', \) and \( T_j \geq T_{j'} \).

This generation of subconstraints carries with it the risk of non-termination, in the case where one type contains itself. Indeed, this is a very real possibility for JavaScript, since every object method has an implicit this parameter which will have the type of the object itself. The solution is to keep track of which types we encounter as we recursively generate subconstraints. If we generate a constraint involving types which we have already encountered, then it is skipped. The soundness of this can be proved by contradiction. A constraint near the top of the path is satisfied if all subconstraints are satisfied. To prove the subconstraints are satisfied, we can assume by induction that the constraint itself is satisfied. If a subconstraint involving the same types appears later on in the path, then we can use this assumption to immediately show that it is satisfied, and hence we can simply skip its generation to avoid infinite recursion.

### 3.5 Functions and Arrays as Objects

Although JSTyper gives functions, arrays and objects distinct types, in practice JavaScript does not make a strong distinction between the three. In JavaScript, everything which is not a primitive type is an object type. This means that arrays and functions can have properties added and removed freely. It could be argued that this is an undesirable behaviour which a type system should exclude, since for example adding a property to an array may well be indicative of a bug. However, there are some indispensable properties of arrays and functions which can only be accessed by dereferencing them as objects. Chief amongst these is the .length property of an array, which is crucial for most tasks involving arrays.

The solution is to treat, as JavaScript does, arrays and functions as if they were simply objects. We can rewrite array types as object types with a special .dereff property representing the array items’ type. This then allows reading and writing properties like any other object. Since array items’ types are covariant with the type of the array itself (as object properties are), we do not need to do anything different for subconstraint generation. Function types are wrapped by an object type with a special .call property containing the original function type. Constraints on the object wrapper will generate a covariant subconstraint on the function type, which can be handled as discussed in the subconstraints section above.
4. Gradual Typing

Although our type checker can verify the safety of the code it analyses, it is very rarely the case that a block of code exists in isolation within the browser. Other pieces of JavaScript may be dynamically loaded from a remote source and included within the same webpage. Browser extensions are typically written in JavaScript too, and either of these may introduce holes in our type system by interaction with our type-checked code. For example, we may define a function \( f \) which expects input of type number, and our static compiler may verify that all visible uses of the function do indeed pass in data of the correct type. However if \( f \) is exposed to unchecked JavaScript, we can no longer be certain that it will only be used in a type-safe way.

On top of this problem, we have previously mentioned that any static type checker must exclude some well behaved programs. Although this is a problem for any language, it is all the more likely to arise in a dynamic language like JavaScript, where the programmer is used to having a certain freedom of expression. A mechanism is thus needed to give programmers freedom to write dynamic code where convenient, while still using a static type checker wherever possible.

To resolve both problems, a programmer can delineate typed portions of their code using “\texttt{jstyper}” directives. They can then annotate their program further with “\texttt{import}” directives, which declare a certain variable as having dynamic behaviour. When read, we cannot rely on an imported variable to have any particular type, and so we must instead insert a dynamic type-check at every point of use, to ensure that its type is as we expect. Listing 5 shows one situation which can only be considered safe if \( x \) returns a boolean value when it first appears, but then returns a numeric value on the next line. Although this behaviour is clearly dynamic, JSTyper is able to perform static analysis, and insert checks to protect the well-typed world from incorrect implementations of \( x \).

4.1 Compiler Implementation

When type-checking a variable identifier, our implementation first checks whether the variable’s name is in the “\texttt{imported}” list. If it is, the current program point is recorded, and a fresh type is returned. After type checking is complete, this fresh type will have been substituted away, and will tell us what type is required at the point of use. The gradual-typing compiler can then replace all such variables with an appropriate wrapper which will dynamically check that the variable has the correct type before returning its value. This wrapper acts as a kind of explicit cast for the variable – from an unknown dynamic type, to some known fixed type. The wrapper will only return values of the specified type, regardless of the actual type of the dynamic variable (though it may throw an error instead of returning). Since the cast is type-preserving, then, the resulting code must be type-safe, and our guarantee that the program will not get stuck is maintained. We may yet generate cast errors, but the well-typed portion of the program cannot be blamed for this error, which must have been caused by the dynamic code returning a value of the incorrect type.

```
4  if (typeof t !== "number")
3       throw new CastError("x is not a number");
2         return t;
1     }());x + 5;
```

Listing 6: An example primitive wrapper

Although it may appear that we have not gained anything by replacing type errors with cast errors, it must be remembered that, in JavaScript, an explicit type error may never be thrown. Instead, values of incorrect types are silently coerced from one type to another, and the error can propagate quite some distance through the program before being spotted. The guarantee of a visible cast error means that this kind of bug is much more likely to be spotted early in the development process. The additional safety guarantees – that the non-dynamic code cannot be blamed – also restricts the location in which the bug may have arisen, which will help locate and correct it much faster.

4.2 Higher Order Casts

The wrapper for a primitive type is simple enough – simply check the type of the variable and return it if it is correct. When looking at higher-order types, we cannot determine by immediate inspection whether the data has the correct type. Almost by definition, for example, we cannot determine the return type of a dynamic function by looking at it. Object and array types have the same problem, since properties can be defined by separate getter and setter functions, and there is no guarantee that a dynamic getter function will always return the same type of value. We will only be able to know the return type once the function has executed, and a return value has been given.

And so, our wrappers for higher order types do exactly that – they mimic the original function by accepting arguments, passing them on to the inner (dynamic) function, then finally examining the return type. If the return type is itself of a higher order, we once again cannot examine its type directly, so we instead give it another mimic wrapper before returning it.

Unfortunately, by passing on parameters directly to the inner function, we are introducing a new potential hole in our type system. The parameters are well typed, but the inner function is not. If the parameter is a callback, we cannot guarantee that it will be used in a type-safe manner. The solution is to guard the callback with a different kind of wrapper as it enters the untyped world. The guard will check the callback’s parameters (wrapping them in a mimic if necessary), call the callback itself, then pass on the return value into the untyped world (wrapping it with another guard if necessary). If untyped code adds a property to a guarded object, that property is only visible within untyped code.

The implementation itself is fairly straightforward once the wrapper types have been determined. The compiler generates an
AST representation of a call to the function `mimic`, providing as parameters the desired type and the imported variable itself. This is used to replace the variable's node in the original program’s AST. UglifyJS then takes the modified AST and performs the code generation stage of the compiler. Listing 7 shows a sample result. Alongside this, a JavaScript file is included containing the implementation of the `mimic` function. The function is thus called whenever the imported variable is accessed. If the desired type of the variable is a primitive, then a simple type check is performed before returning it. If the variable is of a higher-order type, then additional wrappers are created as described above. An extract of the definition of `mimic` is shown in Listing 8.

```
var y = mimic({
  kind: "object",
  memberTypes: {
    f: {
      kind: "function",
      argTypes: [
        { kind: "abstract" }],
      returnType: {
        kind: "primitive",
        type: "number"}
    }
  }
}, x).f();
```

Listing 7: An example higher order wrapper. Here we ensure that the dynamic variable `x` has a function property `f` which returns a number.

Functions are simple to wrap, as demonstrated in Listing 8. Objects are slightly trickier, but property wrappers can be introduced using getters and setter properties introduced with ECMAScript 5, which are supported in all modern browsers. Array types, which posed no great problem to our type checker, actually present the greatest difficulty to wrap. Getter and setter properties can only be defined for known property names, which means they cannot be used for wrapping assignments to an array, where any natural number can be used. ECMAScript 6 promises a solution to this problem, by introducing Proxy objects which can intercept more native operations, however the proposals have not yet been widely implemented, and since Proxies involve new behaviour of language syntax elements, they cannot be emulated by a polyfill library.

Solutions which are immediately available for use in current browsers would involve frequently polling for changes to the wrapped array, or predefining property wrappers for some constant range of natural numbers. The first approach is unsatisfactory because changes may not be noticed in time to prevent errors, and the second because only arrays of a limited size are supported. Of the two possibilities, we have chosen to implement the second in our prototype compiler, allowing us to maintain the type-safety of array accesses within our supported range. There is a tradeoff to be made against memory usage, since a getter and setter need to be individually initialised for every array index in the range. For our tests, it was sufficient to limit arrays to be smaller than 100 elements, though we believe the limit could be raised without a significant degradation in performance.

This system of guards and mimic wrappers is due to Gray [5, 6], and allows us to control the boundary between the typed and untyped worlds. Unchecked data is given a mimic to ensure it is safe in the well-typed world, and type safe data is given a guard to ensure its safety is not compromised by the unchecked world.

```
function mimic(t, obj) {
  // obj is untrusted, but the context is safe
  switch(t.kind){
    case "function":
      if (typeof obj !== "function")
        throw new CastError(typeof obj + " is not a function");
      // f will be a type-safe version of obj
      return function f() {
        // +1 to count "this"
        if (arguments.length + 1 !== t.argTypes.length)
          throw new CastError("Wrong number \ of parameters");
        var args = [];
        // guard inputs before passing them on
        for (var i; i < arguments.length; i++) {
          args[i] = guard(t.argTypes[i + 1], arguments[i]);
        }
        thisType = guard(t.argTypes[0], f);
        var result = obj.apply(thisType, args);
        // mimic the result before returning it
        return mimic(t.returnType, result);
      };
  }
}
```

Listing 8: A simplified extract from the definition of mimic showing how a function can be wrapped.
4.3 Performance

Inserting extra dynamic checks will inevitably have an impact on the performance of the program. If the program makes frequent use of higher-order data – and the common callback paradigm in JavaScript means this will often be the case – the memory footprint of the program may be significantly increased by the wrappers created.

Although this performance cost will be disappointing, there are ways the problem can be mitigated. It should be noted that there are two main reasons for a variable to be imported into the type safe world. The first reason is because the variable is truly dynamic, and the type of the variable can indeed change at any point. For these, every check is necessary. The second reason to import a variable, however, is that some section of code is inaccessible for static checking (for example if it has been dynamically created or loaded). In this latter case, it may be less wasteful to verify the data’s type once at load time than to mediate every interaction with the type-safe world.

Primitive data is easy to verify using JavaScript’s typeof operator. Objects are not much harder, by analysing each property of the object in turn. Since JSTyper is written in JavaScript, we can perform static analysis of a new function’s code at load time to ensure that it has the expected type. Some scope analysis would also be required to determine which variables within the function are not locally defined. These must have been defined within the function’s closure, and should be treated as dynamic since we cannot access their definition to verify their safety. Finally, an array can be verified by analysing each of its elements in turn. In this way we could perform ahead-of-time type analysis of dynamic data and avoid the overhead of creating mimic wrappers. Clearly this analysis also has a cost and may not always be an improvement, but distinguishing between the two types of ‘dynamic’ variable may be advantageous.

Another point to consider is that the JavaScript interpreter must run this kind of type check at all times anyway. In order to perform a coercion between types, for example, it must first check to see what types the values are. If the interpreter is made aware that part of the code has already been statically verified as type safe, it can avoid making such checks. By protecting the boundaries of the type checked world, we are guaranteeing that it will be free of type errors. Interpreters may in fact be able to use this knowledge to achieve better overall performance than the unchecked code could previously allow.

5. Proof

We have proven type soundness for an older version of JSTyper that did not yet support aliasing or prototypes, and thus did not have effects. We are in the process of preparing an updated proof of soundness and do not anticipate significant alterations to most cases: primarily to merge effects particularly on case splits and to update assignment and function calls to incorporate effects into the types. We use an operational semantics for the typed subset of JavaScript behaviour that JSTyper supports, leaving a proof that only untyped programs can be blamed for errors as future work.

5.1 Operational Semantics

The operational semantics is a small-step inductive relation where a state of computation is described by a configuration triple of the form

\[ (m, s, \theta) \]

m is the program code remaining to be executed, s represents the current scope (the set of variables accessible at this program point); and \( \theta \) represents the heap, where variable values are stored. Most of the rules follow an intuitive understanding of JavaScript’s execution semantics – the result of a subexpression is used to determine the type of the enclosing expression. A few points are worth further discussion.

Assignment

\[
\begin{align*}
\text{assignTarget} & \not\in \text{vRef} \\
(\text{assignTarget} \in s, \theta) & \rightarrow (\text{assignTarget}' \in s, \theta') \quad \text{ASSIGN1} \\
(\text{assignTarget} = e, s, \theta) & \rightarrow (e_1, s, \theta') \quad \text{ASSIGN2} \\
(\text{assignTarget} = e, s, \theta) & \rightarrow (e', s, \theta') \\
(\text{vRef} = e, s, \theta) & \rightarrow (\text{vRef} = e', s, \theta') \\
(\text{vRef} = v, s, \theta) & \rightarrow (v, s, \theta \cup \{\text{addr}(\text{vRef}), s, v\}) \quad \text{ASSIGN3}
\end{align*}
\]

Legal assignment targets are not limited in the ECMAScript specification by syntax, but rather by whether or not expressions resolve to a reference. JSTyper identifies an assignment on the basis of syntax alone. In JSTyper, the only valid left hand side of an assignment is a value reference (vRef), such as an identifier, or an expression which will reduce down to one (assignTarget). The value on the heap may be an object or array pointing to other values, and so x.1 and x[2] are also value references. Although value references share a syntactic similarity to dereference-able expressions, they are distinct and do not reduce down to the value itself, this prevents assignments of the form \( 5 = 6 \). This simplification has the consequence of deeming some valid JavaScript assignments illegal, such as the following expression:

\[
1 \{ \text{function}() \{ \text{return} \}; \} \}.x = 5
\]

Although this is a valid JavaScript assignment according to the ECMAScript specification, it would be difficult to use in our operational semantics, since the object returned by the function has no identifier, and hence no entry in either s or \( \Gamma \). In contrast, there is a direct relation between a value reference and a location in the store, defined by the function \( \text{addr}(\text{vRef}, s, \theta) \).

Function Closures

\[
\begin{align*}
(\text{Func}, s, \theta) & \rightarrow (\text{func}, s, x, \theta) \quad \text{FUNC} \\
a_0, \ldots, a_n \text{ are fresh} \\
\theta_0 = \theta \cup \{ \text{ this } : a_0, x_1 : a_1, \ldots, x_n : a_n \} \\
(\theta_0 \cup \{ \text{ this } : \text{func}, x_1 : \text{func} \}, s, \theta) & \rightarrow (\text{call}, \text{body}[m], s, x_0, \theta) \quad \text{CALLANNON}
\end{align*}
\]

When a function is called in JavaScript, the scope includes the full lexical scope at the point of definition. It is thus insufficient to store the function code – we must also save the current scope when a definition is encountered, so that this scope can be used when the function is called. The rule Func handles this storage by converting a function expression into a function closure, which contains both the function code and the scope. The rule CallAnnOn demonstrates an example of the point of use of a closure. It is converted to an in-execution function body denoted by \( \text{@body}[m] \). Fresh heap locations are chosen for the function parameters (including the implicit parameter \text{this}), and these are referenced within the inner scope.
5.2 Proof of Preservation

For this proof, we require a definition of a well-typed store. The intuition behind the definition is that a store is well-typed if all typeable value references exist within a store, and if the stored value itself has the same type as the value reference. We also require a definition of a strength relation (⊆) between type environments. This allows us to verify that the 'output' type environment Γ can still make all the same judgements after a transition.

Definition 8 (Well-typed store).

\[ \Gamma \vdash (s, \theta) \ \text{def} \quad \text{dom}(\Gamma) \subseteq \text{dom}(s) \wedge \]

\[ \Gamma \vdash \text{vRef} : T|c_1 \Gamma' \implies \text{addr}(\text{vRef}, s, \theta) \text{ is defined} \wedge \]

\[ \Gamma \vdash \theta(\text{addr}(\text{vRef}, s, \theta)) : T'|c_1 \Gamma'' \]

\[ T \supseteq T' \]

Definition 10 (Strength).

\[ \Gamma_1 \supseteq \Gamma_2 \ \text{def} \quad \forall (id : T_1) \in \Gamma_1, \ \exists T_2. \ (id : T_2) \in \Gamma_2 \wedge T_1 \supseteq T_2 \]

We also make use of a few useful lemmas.

Lemma 12 If \( \Gamma_1 \supseteq \Gamma_2 \), and \( \Gamma_1 \vdash m \mid c_1 \Gamma_1 \), then \( \Gamma_2 \vdash m \mid c_1 \Gamma_2 \) and \( \Gamma_1 \supseteq \Gamma_2 \).

Lemma 13 The intersection of two types is a supertype to both types.

The proof of Lemma 12 follows from a fairly trivial induction over the structure of the type judgement derivations. The proof of Lemma 13 considers the two well-defined cases according to the definition of ∩. The base case, where \( T_1 \) and \( T_2 \) are not objects, is trivial. When \( T_1 \) and \( T_2 \) are objects, we show that each property present in the intersection type can only have arisen if it was present in both object types, and hence that the intersection must be a supertype.

Theorem 14 (Type preservation for expressions) If \( \Gamma \vdash e : T|c_1 \Gamma' \) and \( \Gamma \vdash (s, \theta) \) and we have some transition \( (e, s, \theta) \rightarrow (e', s', \theta') \), then some \( \gamma \) can augment \( \Gamma \) such that \( \Gamma \cup \gamma \vdash e' : T'|c_1 \Gamma'' \) and \( \Gamma \cup \gamma \vdash (s, \theta) \) and \( \Gamma \supseteq \Gamma'' \).

Theorem 15 (Type preservation for statements) If \( \Gamma \vdash m \mid c_1 \Gamma' \) and \( \Gamma \vdash (s, \theta) \) and we have some transition \( (m, s, \theta) \rightarrow (m', s', \theta') \), then some \( \gamma \) can augment \( \Gamma \) such that \( \Gamma \cup \gamma \vdash m' \mid c_1 \Gamma'' \) and \( \Gamma \cup \gamma \vdash (s, \theta) \) and \( \Gamma \supseteq \Gamma'' \).

Proof. The proof is by rule induction. Theorem 14 induces over all transition rules for expressions, while Theorem 15 induces over transition rules for statements. Take

\[ \Phi(m, s, \theta, m', s', \theta') \ \text{def} \quad \forall \Gamma', C, \Gamma'' . \quad \Gamma \vdash m \mid c_1 \Gamma' \wedge \Gamma \vdash (s, \theta) \implies \]

\[ \exists \gamma, \Gamma'' . \quad \Gamma \cup \gamma \vdash m' \mid c_1 \Gamma'' \wedge \Gamma \cup \gamma \vdash (s', \theta') \wedge \Gamma \supseteq \Gamma'' . \]

where \( C \) and \( C' \) are satisfiable sets of constraints. The transition may cause a change in the store which would not be reflected in \( \Gamma \), so we use the extension \( \Gamma \cup \gamma \). We show that

\[ \Gamma \cup \gamma \vdash m' \mid c_1 \Gamma'' \]

\[ (16) \]

\[ \Gamma \cup \gamma \vdash (s', \theta') \]

\[ (17) \]

\[ \Gamma' \supseteq \Gamma'' . \]

\[ (18) \]

Two interesting cases follow:

Case 1r2

\[ (\text{if}(\text{true})[m_1] \text{else } [m_2], s, \theta) \rightarrow (m_1, s, \theta) \]

\[ \Gamma \vdash e : T_0 \mid c_1 \Gamma_0 \]

\[ \Gamma_0 \vdash m_1 \mid c_1 \Gamma_1 \]

\[ \Gamma_0 \vdash m_2 \mid c_2 \Gamma_2 \]

\[ \Gamma' \vdash \text{merge}(T_0, \Gamma_1, \Gamma_2) \]

\[ \Gamma \vdash (e)[m_1] \text{else } [m_2] \mid c_1\Gamma_0 \cup c_2\Gamma_2 \cup c_1\Gamma_1 \cup c_2\Gamma_2 \]

\[ \text{Iftype1} \]

The only type judgement which could apply to 1r2 is Iftype1. Since \( e = \text{true} \), it must be the case that \( \Gamma = \Gamma_0 \). By assumption, \( \Gamma \vdash (s, \theta) \), and the store is unchanged by the transition, so \( (17) \) is satisfied. Another precondition of Iftype1 is that

\[ \Gamma_0 \vdash m_1 \mid c_1 \Gamma_1 . \]

This gives us \( (16) \) directly, and all that remains to show is that \( \Gamma' \supseteq \Gamma_1 \) for \( (18) \).

\[ \Gamma = \text{merge}(\Gamma_0, \Gamma_1, \Gamma_2) \]

\[ \text{so the elements of } \Gamma' \text{ are the intersection of the corresponding elements in } \Gamma_1 \text{ and } \Gamma_2 \text{, and in particular} \]

\[ \forall (id : T') \in \Gamma' . \quad \exists T_1, \ (id : T_1) \in \Gamma_1, \ (id : T_2) \in \Gamma_2, \]

where \( T' = T_1 \cap T_2 \). Let \( \Gamma' \supseteq \Gamma_1 \). (18) is thus satisfied, and hence \( \Phi(m, s, \theta, m', s', \theta') \).

Case Assign

\[ (\text{vRef}, s, \theta) \in \text{dom(addr)} \]

\[ (\text{vRef} = v, s, \theta) \rightarrow (v, s, \theta \oplus \text{addr}(\text{vRef}, s, \theta) : v) \]

Assign3

\[ T \text{ is fresh} \]

\[ \text{id} \notin \text{dom}(\Gamma) \]

\[ \Gamma \vdash e : T_0 \mid c_1 \Gamma_1 \]

AssnTypeUnd

\[ \Gamma \vdash \text{id} : T_1 \mid c_1 \Gamma_1 \]

AssnType

\[ \Gamma \vdash \text{assignTarget} : T_2 \mid c_1 \Gamma_1 \]

\[ \Gamma \vdash e : T_2 \mid c_1 \Gamma_2 \]

AssnType

\[ \Gamma \vdash \text{id} : T_0 \mid c_1 \Gamma \]

\[ \Gamma \vdash \text{id}.1 \vdash T_1 \mid c_1 \Gamma \]

\[ \Gamma \vdash \text{id}.1.1 \vdash \text{id}.2 \vdash T_2 \mid c_2 \Gamma \]

PropAssignType

\[ \text{Syntactically, there are three rules which could have determined that } e \text{ is typable} \quad \text{AssignType, AssignTypeUnd} \text{ or PropAssignType.} \]

If the rule used was AssignTypeUnd, then \text{vRef} would have to be of the form `id'. The precondition, that \( (\text{vRef}, s, \theta) \in \text{dom(addr)} \), can only hold if \( \text{id} \in \text{dom}(s) \). Since \( \Gamma \vdash (s, \theta) \) this would mean that \( \text{id} \in \Gamma \). This eliminates AssignTypeUnd.

Let \( \theta' = \theta \oplus \text{addr}(\text{vRef}, s, \theta) : v \) in the following discussion.
case (AssignType) An inspection of the possible typing rules for the expression \( \text{vRef} \) (IOTYPE, PROPType or ARRAYType), shows that \( \Gamma \) and \( \Gamma' \) must be equal. We can then deduce (16) from the second precondition of AssignType. We can also determine that \( \Gamma \vdash s, \theta \), since \( s \) is a value, and so (18) is satisfied. By assumption, \( \Gamma \vdash (s, \theta) \), so we just need to show that \( \Gamma \) can handle the addition of the new heap address for (17).

The scope is unchanged, and so \( \text{dom}(\Gamma) \subseteq \text{dom}(s) \) holds. We know that \( \Gamma \vdash \text{vRef} : T_1 \mid C \Gamma \) from the premise of AssignType. We also know that \( \text{addr}(\text{vRef}, s, \theta') \) is well defined, and that \( \theta'(\text{addr}(\text{vRef}, s, \theta')) = v \) from the transition rule Assign3. We have already determined that \( v \) has type \( T_2 \), and the constraints of AssignType give us that \( T_1 \geq T_2 \). Hence \( \Gamma \vdash (s, \theta') \) is satisfied, and \( \Phi(e, s, \theta, e', s', \theta') \) holds.

case (PROPAssigntype) The complication of this case comes from the fact that the output type environment, \( \Gamma_{1} \), is larger than the input, \( \Gamma \). To handle this we do not use an empty \( y \), but instead use \( y = \{ i : \{ i : T \} \} \).

\( \Gamma \vdash y = \Gamma_{1} \), and so (18) is trivially satisfied. We can also show that (16) holds because \( m' \) is a value.

\[ \text{Proof:} \] \[ \Phi(\Gamma, e, T, C, \Gamma') \equiv \forall s, \theta . \quad (\Gamma \vdash (s, \theta)) \implies m = v \lor \exists m', e', \theta' . \quad (m, s, \theta) \rightarrow (m', e', s', \theta') \]

For statements, take

\[ \Phi(\Gamma, e, T, C, \Gamma') \equiv \forall s, \theta . \quad (\Gamma \vdash (s, \theta)) \implies m = v \lor \exists m', e', \theta' . \quad (m, s, \theta) \rightarrow (m', e', s', \theta') \]

We show that for all \( \Gamma, m, T, C, \Gamma' \), if \( \Gamma \vdash m \mid C \Gamma \), then \( \Phi(\Gamma, m, C, \Gamma') \) is satisfied by considering the last rule used in the derivation of \( \Gamma \vdash m \mid C \Gamma \). In all of the cases, we consider an arbitrary store \( (s, \theta) \), and assuming that it is well typed under \( \Gamma \). This gives us the left hand side of \( \Phi(\Gamma, m, C, \Gamma') \), and it remains to prove that \( m = v \lor m = \text{return } v; m' \lor \exists m', e', s', \theta' . \quad (m, s, \theta) \rightarrow (m', e', s', \theta') \).

Here there is one interesting case:

Case AssignType

\[ \text{assignTarget}(s, \theta) \rightarrow \text{assignTarget'}(s', \theta') \quad \text{Assign1} \]

\[ \text{assignTarget} = e_1, \theta \rightarrow \text{assignTarget'} = e_1, \theta' \quad \text{Assign2} \]

\[ \text{vRef} \in e, s, \theta \rightarrow \text{vRef} = e', s', \theta' \quad \text{Assign3} \]

Using the induction hypothesis, we can deduce that \( e_0 \) and \( e_1 \) are both either values or further reducible. Syntactically, however, \( e_0 \) cannot be a value, because the left hand side of an assignment must be some form of assignTarget.

case \( e_0 \) reduces further and is not of the form vRef. The precondition for rule Assign1 is thus satisfied, and \( e \) itself reduces further.

case \( e_0 \) is of the form vRef and \( e_1 \) reduces further. The precondition for rule Assign2 is thus satisfied, and \( e \) itself reduces further.

case \( e_0 \) is of the form vRef and \( e_1 \) is a value. The only rule we can use to reduce \( e \) here is Assign3. For this to be possible, we need to show that \( \Phi(e_0, s, \theta) \rightarrow \text{dom}(\text{addr}) \). We have assumed that the store \( (s, \theta) \) is well-typed. Since \( e_0 \) is typable, we can deduce that \( \text{addr}(e_0, s, \theta) \) is indeed well-defined. Thus Assign is applicable, and \( e \) reduces further.

\[ \Box \]
6. In Practice

JSTyper has successfully type checked several programs from the SunSpider benchmark suite. The main factor preventing tests on the full range is the availability of JSTyper-compatible type definitions for built-in objects. For example, many tests involve manipulation of the Document Object Model representing a web page, but the type definition for these objects is not available to JSTyper, and simply importing all references would leave very little content to actually test within the benchmark.

The SunSpider test suite is designed to avoid microbenchmarks, and aims to reflect real-world use of JavaScript. Our findings were that in fact, very few tests presented truly dynamic behaviour, and largely behave statically. That such code is touted as representative of production JavaScript gives us further confidence that in practice, most programmers obey a static typing discipline even when the language itself does not enforce it.

However, the SunSpider tests largely involved numerical calculations, which failed to take full advantage of JSTyper’s ability to handle higher order types. As such, we tested JSTyper using our own more elaborate scripts, involving deep objects with property addition, object methods and inheritance using the function prototype. All static type errors in our tests were correctly identified, and type information generated such that the correct wrappers could be generated for dynamic variables. Although we believe these tests to also be representative of ‘real-world’ JavaScript, further work will involve testing code which is currently in production use. An example of a test involving the import of the functional programming library underscore.js is included in Listing 9 Without JSTyper’s gradual typing compilation, this program would concatenate all prices together, resulting in the output ‘012323’. This is unlikely to be what the programmer intended, as indicated by the addition of a string with a number. JSTyper does not have access to the definition of _.reduce, and yet by inserting wrappers at strategic points, we can detect the error. Indeed, when the code is compiled and executed, an error is thrown instead.

```javascript
// jstyper start
// jstyper import _

var total = _.reduce(_.map([1,2,3,2,3],
function (units) {
  var order = {
    units: units
  }
  order.price = units * 5;
  return order;
}),
function (d1, d2) {
  return d1 + d2.price;
},
// Deliberate bug - 0 should not be a string
"0"
);
// jstyper end
```

Listing 9: An example of safe functional programming in JavaScript with JSTyper and underscore.js

7. Conclusions

JSTyper provides a static type system for JavaScript without requiring annotations and still supporting the bulk of idiomatic JavaScript code. We have made some compromises over flexibility and static checks, with regard to highly overloaded functions such as + as one would expect in a type system. We continue to support interaction with standard JavaScript and these functions through gradual typing support.

Next steps with JSTyper are to explore an integration with asm.js support, to cause our type system compiler to generate asm.js annotated functions where possible to provide performance enhancements without additional programmer overhead; and to extend the safety guarantees provided within that subset of JavaScript to as much of the JavaScript world as JSTyper can check, with runtime guards to preserve that safety.

References


